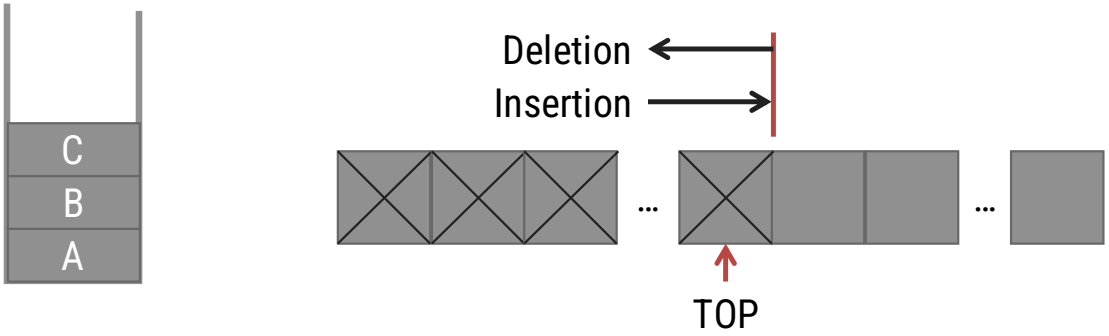
Stack

- A linear list which allows insertion and deletion of an element at one end only is called ***stack***.
- The insertion operation is called as ***PUSH*** and deletion operation as ***POP***.
- The most accessible elements in stack is known as ***top***.
- The elements can only be removed in the opposite orders from that in which they were added to the stack.
- Such a linear list is referred to as a ***LIFO (Last In First Out)*** list.



Stack Cont…

- A pointer TOP keeps track of the top element in the stack.
- Initially, when the ***stack is empty***, **TOP** has a value of ***"zero"***.
- Each time a **new element is inserted** in the stack, the pointer is ***incremented by "one"*** before, the element is placed on the stack.
- The pointer is ***decremented by "one"*** each time a **deletion** is made from the stack.

Applications of Stack

- Recursion
- Keeping track of function calls
- Evaluation of expressions
- Reversing characters
- Servicing hardware interrupts
- Solving combinatorial problems using backtracking
- Expression Conversion (Infix to Postfix, Infix to Prefix)
- Game Playing (Chess)
- Microsoft Word (Undo / Redo)
- Compiler – Parsing syntax & expression
- Finding paths

Procedure : PUSH (S, TOP, X)

- This procedure inserts an element **X** to the top of a stack.
- Stack is represented by a vector **S** containing **N** elements.
- A pointer **TOP** represents the top element in the stack.

```
1. [Check for stack overflow]
      If    TOP ≥ N
      Then  write ('STACK OVERFLOW')
            Return
2. [Increment TOP]
      TOP ← TOP + 1
3. [Insert Element]
      S[TOP] ← X
4. [Finished]
      Return
```

Stack is empty, TOP = 0, N=3

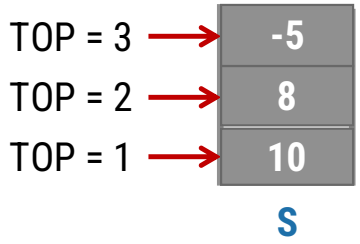
PUSH(S, TOP, 10)

PUSH(S, TOP, 8)

PUSH(S, TOP, -5)

PUSH(S, TOP, 6)

Overflow



Function : POP (S, TOP)

- This function ***removes & returns*** the top element from a stack.
- Stack is represented by a vector **S** containing **N** elements.
- A pointer **TOP** represents the top element in the stack.

```
1. [Check for stack underflow]
      If    TOP = 0
      Then  write ('STACK UNDERFLOW')
            Return (0)
2. [Decrement TOP]
      TOP ← TOP - 1
3. [Return former top element of
   stack]
      Return(S[TOP + 1])
```

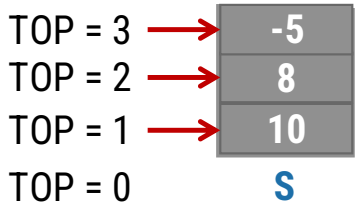
POP(S, TOP)

POP(S, TOP)

POP(S, TOP)

POP(S, TOP)

Underflow



Function : PEEP (S, TOP, I)

- This function returns the value of the **Ith** element from the **TOP** of the stack. The element is not deleted by this function.
- Stack is represented by a vector **S** containing **N** elements.

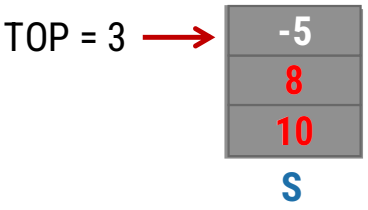
```
1. [Check for stack underflow]
      If    TOP-I+1 ≤ 0
      Then  write ('STACK UNDERFLOW')
            Return (0)
2. [Return Ith element from top
    of the stack]
      Return(S[TOP-I+1])
```

PEEP (S, TOP, 2)

PEEP (S, TOP, 3)

PEEP (S, TOP, 4)

Underflow



PROCEDURE : CHANGE (S, TOP, X, I)

- This procedure changes the value of the **Ith** element from the top of the stack to **X**.
- Stack is represented by a vector **S** containing **N** elements.

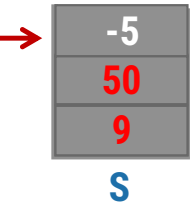
```
1. [Check for stack underflow]
      If    TOP-I+1 ≤ 0
      Then  write ('STACK UNDERFLOW')
            Return
2. [Change Ith element from top
    of the stack]
      S[TOP-I+1] ← X
3. [Finished]
      Return
```

CHANGE (S, TOP, 50, 2)

CHANGE (S, TOP, 9, 3)

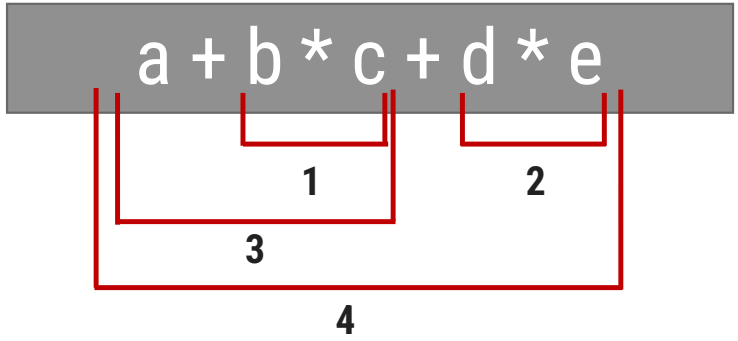
CHANGE (S, TOP, 25, 8)

Underflow



Polish Expression & their Compilation

- Evaluating Infix Expression



- A **repeated scanning** **from left to right is needed** as operators appears inside the operands.
- ***Repeated scanning is avoided*** if the **infix expression** is first **converted** to an equivalent parenthesis free ***prefix or suffix (postfix) expression***.
- **Prefix Expression: Operator**, Operand, Operand
- **Postfix Expression:** Operand, Operand, **Operator**

Polish Notation

- This type of notation is known **Lukasiewicz Notation** or **Polish Notation** or **Reverse Polish Notation** due to Polish logician *Jan Lukasiewicz*.
- In both **prefix** and **postfix** equivalents of an infix expression, the ***variables are in same relative position***.
- The expressions in postfix or prefix form are ***parenthesis free*** and operators are rearranged according to rules of precedence for operators.

Polish Notation

Sr.	Infix	Postfix	Prefix
1	a	a	a
2	a + b	a b +	+ a b
3	a + b + c	a b + c +	+ + a b c
4	a + (b + c)	a b c + +	+ a + b c
5	a + (b * c)	a b c * +	+a * b c
6	a * (b + c)	a b c + *	* a + b c
7	a * b * c	a b *c*	** a b c

Finding Rank of any Expression

E = (A + B * C / D - E + F / G / (H + I))

Note: R = *Rank*, Rank of Variable = *1*, Rank of binary operators = *-1*

Rank (E) = R(A) + R(+) + R(B) + R(*) + R(C) + R (/) + R(D) + R(-) + R(E) + R(+) + R(F) + R(/) + R(G) + R(/) + R(H) + R(+) + R(I)

Rank (E) = 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1

Rank (E) = 1

Any Expression is valid if Rank of that expression is 1

Convert Infix to Postfix Expression

Symbol	Input precedence function (F)	Stack precedence function (G)	Rank function (R)
+, -	1	2	-1
***, /**	3	4	-1
^	6	5	-1
Variables	7	8	1
(	9	0	-
)	0	-	-

Algorithm : REVPOL

- Given an input string **INFIX** containing an infix expression which has been padded on the right with **')'**.
- This algorithm ***converts INFIX into reverse polish*** and places the result in the string **POLISH**.
- All symbols have precedence value given by the table.
- Stack is represented by a vector **S, TOP** denotes the top of the stack, algorithm **PUSH** and **POP** are used for stack manipulation.
- Function **NEXTCHAR** returns the next symbol in given input string.
- The integer variable **RANK** contains the rank of expression.
- The string variable **TEMP** is used for temporary storage purpose.

```
1. [Initialize Stack]
    TOP ← 1
    S[TOP] ← '('
2. [Initialize output string and rank count]
    POLISH ← ''
    RANK ← 0
3. [Get first input symbol]
    NEXT ← NEXTCHAR(INFIX)
4. [Translate the infix expression]
    Repeat thru step 7
       while NEXT != ' '
```

Symbol	IPF (F)	SPF (G)	RF (R)
+, -	1	2	-1
*, /	3	4	-1
^	6	5	-1
Variables	7	8	1
(	9	0	-
)	0	-	-

```
5. [Remove symbols with greater precedence
   from stack]
    IF   TOP < 1
    Then write ('INVALID')
         EXIT
    Repeat while G(S[TOP]) > F(NEXT)
        TEMP ← POP (S, TOP)
        POLISH ← POLISH O TEMP
        RANK ← RANK + R(TEMP)
        IF   RANK <1
        Then write ('INVALID')
             EXIT
6. [Are there matching parentheses]
   IF   G(S[TOP]) != F(NEXT)
   Then call PUSH (S,TOP, NEXT)
   Else POP (S,TOP)
7. [Get next symbol]
   NEXT ← NEXTCHAR(INFIX)
8. [Is the expression valid]
    IF   TOP != 0 OR RANK != 1
    Then write ('INVALID')
    Else write ('VALID')
```

(a + b ^ c ^ d) * (e + f / d))

NEXT

1. [Initialize Stack]
   TOP ← 1
   S[TOP] ← ‘(’
2. [Initialize output string and rank count]
   POLISH ← ‘’
   RANK ← 0
3. [Get first input symbol]
   NEXT ← NEXTCHAR(INFIX)

Input Symbol	Content of stack	Reverse polish expression	Rank
	(		0
(			

(a + b ^ c ^ d) * (e + f / d))

NEXT

```
4. [Translate the infix expression]
    Repeat thru step 7
          while NEXT!= ‘ ‘

5. [Remove symbols with greater precedence from
   stack]
   IF    TOP < 1
   Then write (‘INVALID’)
        EXIT
   Repeat while G(S[TOP]) > F(NEXT)
       TEMP ← POP (S, TOP)
       POLISH ← POLISH O TEMP
       RANK ← RANK + R(TEMP)
       IF          RANK <1
       Then write (‘INVALID’)
            EXIT

6. [Are there matching parentheses]
    IF    G(S[TOP]) != F(NEXT)
    Then call PUSH (S,TOP, NEXT)
    Else POP (S,TOP)

7. [Get next symbol]
    NEXT ← NEXTCHAR(INFIX)
```

Input Symbol	Content of stack	Reverse polish expression	Rank
	(		0
(	((		0
a	((a		0
+	((+	a	1
b	((+b	a	1
^	((+^	ab	2
c	((+^c	ab	2
^	((+^^	abc	3
d	((+^^d	abc	3
)	((	abcd^^+	1

Perform following operations

- **Convert** the following **infix** expression to **postfix**. Show the stack contents.
  - A$B-C*D+E$F/G
  - A + B – C * D * E $ F $ G
  - a+b*c-d/e*h
  - ((a+b^c^d)*(e+f/d))
- Convert the following **infix** expression to **prefix**.
  - A + B – C * D * E $ F $ G

General Infix-to-Postfix Conversion

Create an empty stack called stack for keeping operators. Create an empty list for output.

Read the character list from left to right and perform following steps

1. If the **character** is an **operand (Variable)**, **append** it to the **end** of the **output** list
2. If the **character** is a **left parenthesis '('**, **push** it on the **stack**
3. If the **character** is a **right parenthesis ')'**, **pop** the **stack until** the corresponding **left parenthesis ')'** is **removed**. **Append** each **operator** to the **end of the output** list.
4. If the token is an **operator**, ***, /, +,** or **-**, **push** it on the **stack**. However, **first remove any operators** already on the **stack** that **have higher or equal precedence** and **append them to the output** list.

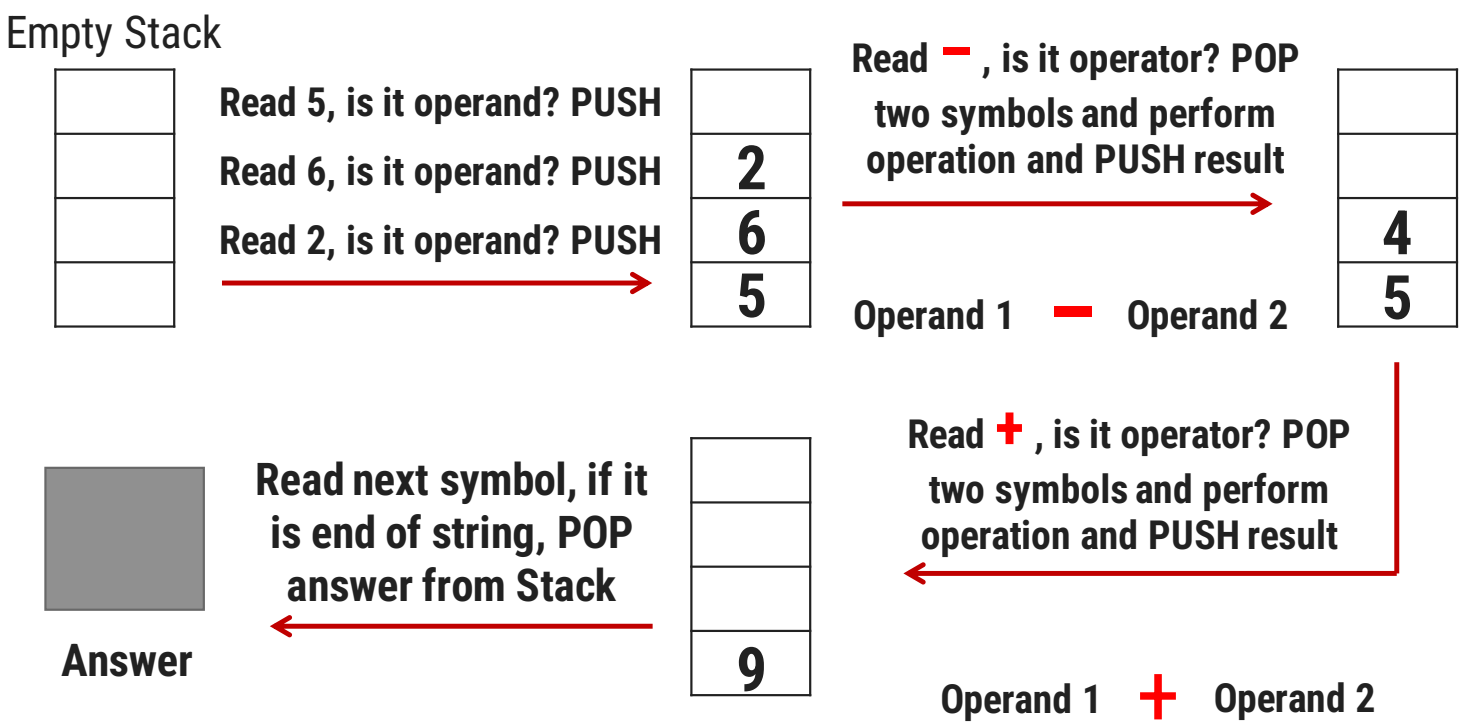
(a + b ^ c ^ d) * (e + f / d)

Evaluation of postfix expression

- Each **operator** in **postfix** string **refers** to the *previous two operands* in the string.
- Each time we **read** an **operand**, we **PUSH** it onto **Stack**.
- When we reach an **operator**, its **operands** will be **top two elements** on the stack.
- We can then **POP** these two elements, perform the indicated operation on them and PUSH the result on the stack so that it will available for use as an operand for the next operator.

Evaluation of postfix expression

Evaluate Expression: 5 6 2 - +



Algorithm: EVALUATE_POSTFIX

- Given an input string **POSTFIX** representing postfix expression.
- This algorithm evaluates postfix expression and put the result into variable **VALUE**.
- Stack is represented by a vector **S, TOP** denotes the top of the stack, Algorithm **PUSH** and **POP** are used for stack manipulation.
- Function **NEXTCHAR** returns the next symbol in given input string.
- **OPERAND1**, **OPERAND2** and **TEMP** are used for temporary variables
- **PERFORM_OPERATION** is a function which performs required operation on **OPERAND1** & **OPERAND2**.

Algorithm: EVALUATE_POSTFIX

```
1. [Initialize Stack]
    TOP ← 0
    VALUE ← 0

2. [Evaluate the postfix expression]
   Repeat until last character
      TEMP ← NEXTCHAR (POSTFIX)
      If TEMP is DIGIT
      Then PUSH (S, TOP, TEMP)
      Else OPERAND2 ← POP (S, TOP)
           OPERAND1 ← POP (S, TOP)
           VALUE ← PERFORM_OPERATION(OPERAND1, OPERAND2, TEMP)
           PUSH (S, POP, VALUE)

3. [Return answer from stack]
   Return (POP (S, TOP))
```

Evaluation of postfix expression

Evaluate Expression: 5 4 6 + * 4 9 3 / + *

Evaluate Expression: 7 5 2 + * 4 1 1 + / -

Evaluate Expression: 12, 7, 3, -, /, 2, 1, 5, +, *, +

Algorithm: EVALUATE_PREFIX

- Given an input string **PREFIX** representing prefix expression.
- This algorithm evaluates prefix expression and put the result into variable **VALUE**.
- Stack is represented by a vector **S, TOP** denotes the top of the stack, Algorithm **PUSH** and **POP** are used for stack manipulation.
- Function **NEXTCHAR** returns the next symbol in given input string.
- **OPERAND1**, **OPERAND2** and **TEMP** are used for temporary variables
- **PERFORM_OPERATION** is a function which performs required operation on **OPERAND1** & **OPERAND2**.

Algorithm: EVALUATE_PREFIX

```
1. [Initialize Stack]
    TOP ← 0
    VALUE ← 0

2. [Evaluate the prefix expression]
   Repeat from last character up to first
       TEMP ← NEXTCHAR (PREFIX)
       If   TEMP is DIGIT
       Then PUSH (S, TOP, TEMP)
       Else OPERAND1 ← POP (S, TOP)
            OPERAND2 ← POP (S, TOP)
            VALUE ← PERFORM_OPERATION(OPERAND1, OPERAND2, TEMP)
            PUSH (S, POP, VALUE)

3. [Return answer from stack]
   Return (POP (S, TOP))
```

Recursion

A **procedure** that contains a **procedure call to itself** or a procedure **call to second procedure** which eventually **causes the first procedure to be called** is known as **recursive procedure**.

Two important conditions for any recursive procedure	
1	Each time a procedure calls itself it must be nearer in some sense to a solution.
2	There must be a decision criterion for stopping the process or computation.

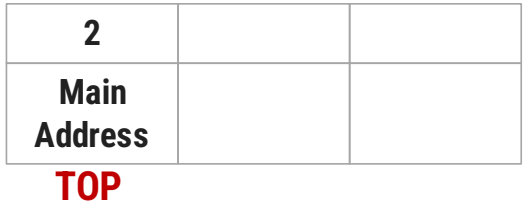
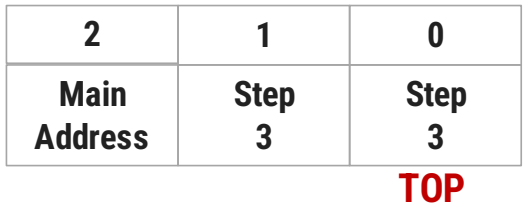
Algorithm to find factorial using recursion

- Given integer number **N**
- This algorithm **computes factorial of N**.
- **Stack S** is used to store an activation record associated with each recursive call.
- **TOP** is a pointer to the top element of stack S.
- Each **activation record contains** the current value of **N** and the current return address **RET_ADDE**.
- **TEMP_REC** is also a record which contains two variables **PARAM** & **ADDRESS**.
- **Initially** return address is set to the **main calling address**. PARAM is set to initial value N.

Algorithm: FACTORIAL

```
1. [Save N and return Address]
      CALL PUSH (S, TOP, TEMP_REC)
2. [Is the base criterion found?]
      If     N=0
      then   FACTORIAL ← 1
             GO TO Step 4
      Else   PARAM ← N-1
             ADDRESS ← Step 3
             GO TO Step 1
3. [Calculate N!]
      FACTORIAL← N * FACTORIAL
4. [Restore previous N and return address]
      TEMP_REC ← POP(S,TOP)
      (i.e. PARAM ← N, ADDRESS ← RET_ADDR)
      GO TO ADDRESS
```

Trace of Algorithm FACTORIAL, N=2

Level Number	Description	Stack Content
Enter Level 1 (main call)	Step 1: PUSH (S,0,(N=2, main address)) Step 2: N!=0 PARAM ← N-1 (1), ADDR ← Step 3	
Enter Level 2 (first recursive call)	Step 1: PUSH (S,1,(N=1, step 3)) Step 2: N!=0 PARAM ← N-1 (3), ADDR ← Step 3	
Enter Level 3 (second recursive call)	Step 1: PUSH (S,2,(N=0, step 3)) Step 2: N=0 FACTORIAL ← 1 Step 4: POP(A,3) GO TO Step 3	 

Trace of Algorithm FACTORIAL, N=2

Level Number	Description	Stack Content
Return to Level 2	Step 3: FACTORIAL ← 1*1 Step 4: POP (A,2) GO TO Step 3	**2** **Main Address** **TOP**
Return to Level 1	Step 3: FACTORIAL ← 2*1 Step 4: POP (A,1) GO TO Main Address **TOP = 0**	